Optimizing Quantum Circuit Synthesis for Permutations Using Recursion

Cynthia Chen
Caltech
Pasadena, USA
cchen6@caltech.edu

Bruno Schmitt
EPFL
Lausanne, Switzerland
bruno.schmitt@epfl.ch

Helena Zhang, Lev S. Bishop, Ali Javadi-Abhar
IBM Quantum
Yorktown Heights, NY, USA
{helena.zhang,ali.javadi}@ibm.com

ABSTRACT

We describe a family of recursive methods for the synthesis of qubit permutations on quantum computers with limited qubit connectivity. Two objectives are of importance: circuit size and depth. In each case we combine a scalable heuristic with a non-scalable, yet exact, synthesis.

ACM Reference Format:
Cynthia Chen, Bruno Schmitt, and Helena Zhang, Lev S. Bishop, Ali Javadi-Abhar. 2022. Optimizing Quantum Circuit Synthesis for Permutations Using Recursion. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530654

1 INTRODUCTION

There is a strong belief that quantum computers will be able to solve certain problems beyond the reach of classical computers [3, 14]. Typically, researchers describe quantum algorithms in high levels of abstraction: from purely mathematical description to high-level quantum circuits—a sequence of operators acting on qubits. While these high-level circuits assume all-to-all connectivity, in practice, the qubits in most quantum hardware are not fully connected due to noise and interconnect challenges. Therefore, not every qubit pair can participate in the same gate operation. These connectivity restrictions are known as coupling constraints.

The task of finding a mapping from virtual instructions to allowed physical instructions is known as quantum circuit mapping. Completing this task is not always possible without applying additional operators to the circuit. These additional operators enable the execution of gates on non-adjacent qubits by permuting their place in the coupling graph. Their cost, however, can dominate the total cost for many applications. This work focuses on families of circuits *permute* qubits on a coupling graph. Also, these permutation circuits appear prominently in quantum computing benchmarks [6].

The size and depth of a circuit are two important metrics to evaluate the quality of a synthesis process. Size refers to the number of gates and is relevant since each gate is noisy, and many gates can cause errors to accumulate. Depth refers to the number of timesteps (or layers) in the circuit. Since qubits have limited coherence, they retain information for a short time. Hence, deep circuits can get overwhelmed by noise. Note that there is often a tradeoff between size and depth optimization.

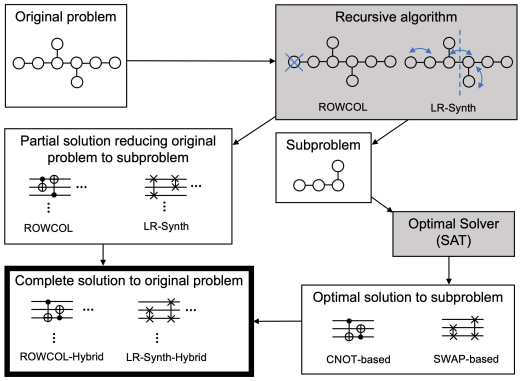

Figure 1: High-level view of our hybrid approach for synthesizing permutations on arbitrary topologies. It takes as input a graph and the desired permutation. These are given to either ROWCOL [15] or LR-Synth, depending on whether CNOT or SWAP is used as primitive. Both generate a partial solution that is combined with an optimal solution synthesized by a SAT-based technique.

An intuitive procedure for the physical synthesis of permutations is to perform SWAP operations between pairs of qubits. The size-optimizing version of this problem is known as token swapping [16], and the depth-optimizing version is known as routing via matchings [1]. Finding optimal solutions to either of these problems takes exponential time. While swapping qubits is the most common model employed to solve this problem due to its analogy to graph algorithms, we can improve by leveraging fundamental quantum gates such as CNOT. Since permutations are a special class of linear functions over $\mathbb{F}_2$, they are computable using a series of XOR operations (i.e., CNOTs).

Broadly, existing methods for synthesizing permutation circuits either optimize for size or depth, using either SWAPs or CNOTs. Some methods are exact but non-scalable, some are tailored to certain topologies, and some are general-purpose heuristics. In Tables 1 and 2 we summarize prior literature, ordered chronologically and separated by whether they optimize for size or depth.

1.1 Our Contributions

In this work, we utilize recursive heuristics for the synthesis of permutations. They have two key advantages: first, they reduce the problem at each stage, making high-quality solutions easier to find and can be used in conjunction with optimal methods in the inner

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530654

	Description	Primitive Op.	Topologies	Size bounds
Patel ('08) [10]	Grouped Gaussian elimination	CNOT	Fully connected	$O\left(\frac{n^2}{\log n}\right)$
Kissinger ('19) [8], Gheorghiu ('20) [7]	Gaussian elimination with Steiner trees	CNOT	Any	$O(n^2)$
Wu ('19) [15]	Recursive elimination of ROW_i/COL_i	CNOT	Any	$2n^2$
Schmitt ('20) [12]	A* search with admissible heuristic	SWAP	Any	Optimal (small n)
Ours	SAT formulation	CNOT	Any	Optimal (small n)

Table 1: Summary of size-optimizing methods. Bounds are in terms of the operation the method is based on. 1 **SWAP** = 3 **CNOTs.**

	Description	Primitive Op.	Topologies	Depth bounds
Alon ('94)[1]	Graph matchings	SWAP	Tree, cartesian prod., etc.	$3n$
Zhang ('97) [17]	Caterpillar partition and matchings.	SWAP	Tree	$\frac{3}{2}n + O(\log n)$
Kutin ('07) [9]	Odd-even transposition sort (+ manual for specific permutations)	CNOT	Line	n
Wu ('19) [15]	Using n^2 ancillas	CNOT	2D grid	$O(\log n)$
Schmitt ('20) [12]	SAT formulation	SWAP	Any	Optimal (small n)
de Brugiere ('21) [4]	Divide and conquer	CNOT	Fully connected	$\frac{4}{3}n + 8\log_2(n)$
Bapat ('21) [2]	Divide and conquer	Reversal	Any	$O(k^2) + \frac{2}{3}r$
Ours	SAT formulation	CNOT	Any	Optimal (small n)
Ours	Divide and conquer	SWAP	Any	$2n + 2\log n$

Table 2: Summary of depth-optimizing methods. Bounds are in terms of operation the method is based on. 1 **SWAP** = 3 **CNOTs.**

stages of recursion (Figure 1). Second, with depth optimization, it is important to parallelize the circuit as much as possible, and recursive methods can be parallelized at each stage. Our approach scales to thousands of qubits and applies to any qubit connectivity topology, and yields better results than state-of-the-art approaches.

First, we propose a SAT encoding of the problem, which can find circuits with optimal size or depth using CNOTs. This technique improves over prior SWAP-based optimal solvers [12] and can be used to synthesize any linear function, not just permutations.

Equipped with optimal solvers (hereafter referred to as *CNOT-size-optimal*, *CNOT-depth-optimal*, *SWAP-size-optimal*, *SWAP-depth-optimal*), we use them in recursive heuristics and scale to much larger problem instances. For size, we rely on a modified version of the ROWCOL heuristic by Wu et al. [15], which outperforms other size-oriented algorithms and lends itself to recursion. No such algorithm exists for depth, so we propose *LR-Synth*, a novel depth-optimizing divide-and-conquer heuristic for general graphs.

ROWCOL is a recursive algorithm that reduces the problem by removing one non-cut vertex/qubit from the connectivity graph at a time. We observe that the order of vertex elimination is important and optimize for that. Combining ROWCOL with CNOT-size-optimal for the inner stages of recursion yields our hybrid algorithm, which significantly outperforms prior work.

Our second recursive algorithm, LR-Synth, is a novel algorithm that routes qubits to the correct half of the topology in each step then recurses on the left and right subgraphs of the topology. Since we can always divide a topology into two parts, an advantage of LR-Synth is that it can be applied to any topology connectivity, addressing a current limitation of depth optimization approaches. We benchmark LR-Synth against Schmitt's SAT-based token swapping solver [12], which solves for optimal SWAP depth. We also benchmark LR-Synth against state-of-the-art heuristics that exist for specific topologies (paths, trees, and lattices). Overall we find that LR-Synth achieves close to optimal circuit depth while being much more scalable than exact solvers and applicable to any topology. We emphasize the generality of the method since many realistic quantum architectures are not built using regular topologies.

We note that there are two gaps in prior work which we fill. First, while exact solvers exist for size-optimal and depth-optimal synthesis based on SWAPs [12] (and available in software package Tweedledum [11]), no CNOT-optimal synthesizers are known. Since each SWAP is equivalent to three CNOTs, direct usage of CNOTs in synthesis can yield better circuits. Second, to our knowledge, there are no scalable heuristics for depth minimization on general topologies (with one recent exception [2], which assumes the existence of fast reversal operations that may not be available on some architectures).

Finally, we use our exact algorithm to disprove a 15-year-old conjecture by Kutin et al. [9] that reversal is at least as depth-intensive to synthesize with CNOTs as any permutation on a path.

2 OPTIMAL SAT-BASED ALGORITHM FOR CNOT CIRCUITS

We formulate the problem of finding an optimal-depth circuit for a linear matrix representing a reversible function as instances of the Boolean satisfiability problem. In our encoding, we use two kinds of variables. Matrix variables, $m^d_{i,k}$, which indicate whether a matrix entry (i, k) is 0 or 1 at depth d; and CNOT gate variables, $g^d_{c\to t}$, which indicate that a CNOT between qubits c and t took place at depth d. For example a 3-qubit linear function synthesis

would be encoded as such, where a 3×3 Boolean matrix represents the linear function over 3 bits, and a CNOT application transforms the matrix by XOR-ing two rows.

$$\begin{bmatrix} m^d_{0,0} & m^d_{0,1} & m^d_{0,2} \\ m^d_{1,0} & m^d_{1,1} & m^d_{1,2} \\ m^d_{2,0} & m^d_{2,1} & m^d_{2,2} \end{bmatrix} \xrightarrow[g^d_{0\to 2}]{} \begin{bmatrix} m^d_{0,0} & m^d_{0,1} & m^d_{0,2} \\ m^d_{1,0} & m^d_{1,1} & m^d_{1,2} \\ m^d_{0,0} \oplus m^d_{2,0} & m^d_{0,1} \oplus m^d_{2,1} & m^d_{0,2} \oplus m^d_{2,2} \end{bmatrix}$$

Our encoding uses four different types of clauses to constrain the problem such that the solution corresponds to a valid linear reversible circuit:

- **C1.** Each depth that does not hold the target transformation must have at least one CNOT.
$$\forall\{d : \exists(d+1)\}, \quad \sum_{(c,t)\in E} g^d_{c\to t} + g^d_{t\to c} \geq 1.$$
- **C2.** At each depth that has at least one CNOT, each qubit can only be involved in one CNOT.
$$\forall\{d : \exists(d+1)\}, \forall c \in V, \quad \sum_{t\in\delta(c)} g^d_{c\to t} + g^d_{t\to c} = 1.$$
where $\delta(c)$ is the set of qubits adjacent to c.
- **C3.** If at depth d the variable indicating a $CNOT(i,k)$ is true, then all elements of the k-th row at depth $d+1$ must be XOR-ed between the element and its corresponding element in the i-th row at the previous depth d.
$$g^d_{c\to t} \implies \bigwedge_j \left(m^{d+1}_{t,j} = m^d_{t,j} \oplus m^d_{c,j}\right)$$
- **C4.** If at depths d and $d+1$ a matrix entry in the i-th row has different values, then exactly one of the CNOT variables that has i as target must be true.
$$m^{d+1}_{t,j} \neq m^d_{t,j} \implies \sum_{c\in\delta(t)} g^d_{c\to t} = 1$$

The SAT solver only answers whether a given formula is satisfiable or unsatisfiable. Therefore, we need to translate our optimization objective into a series of queries to the SAT solver. In this case, each query "asks" the solver if there exists a circuit that implements the desired transformation using a specified depth. Our implementation incrementally solves the problem: first, we build a formula that encodes a solution with a specified depth using the above constraints; then, if the formula is unsatisfiable, we increment the depth by adding new variables and constraints. We keep incrementing the depth until we find a satisfiable formula to decode and build a linear reversible circuit.

We can use a slightly different encoding to find reversible CNOT circuits with optimal size. In such a case, the only difference lies in the first type of constraints: Instead of requiring depths to have at least one CNOT, we restrict the number to exactly one.

As we will show, these SAT-based solutions can be applicable with scalable heuristics to improve their quality, but, they are also useful on their own. For example, we studied CNOT-depth-optimal solutions for permutations on a path, which was the focus of Kutin et al.'s paper [9]. They conjectured that reversals are at least as hard as any other permutation, and that reversals are synthesizable with depth $2n+2$ CNOTs. However, by solving for all instances of 8-qubit permutations, we found a specific permutation that required depth $2n+3 = 19$, as shown in Figure 2, thus disproving the conjecture.

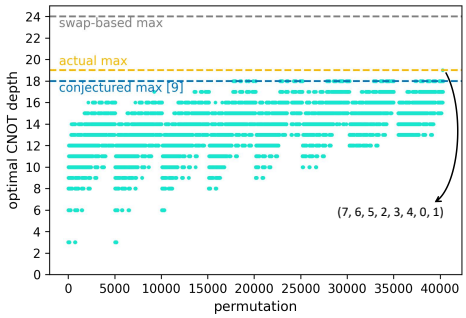

Figure 2: Optimal depth of all permutations on an 8Q path synthesized by CNOT-depth-optimal.

3 SIZE OPTIMIZATION: ROWCOL-HYBRID

3.1 Algorithm Description

We propose and implement a hybrid CNOT circuit synthesis algorithm that combines a modification on Wu's ROWCOL algorithm [15] with optimal SAT-based methods, a generalizable approach (Figure 1).

In [15], non-cut vertices are iteratively removed from the graph, though choosing the order of removing non-cut vertices is not discussed. We find that removal order has a non-trivial effect on circuit size and depth (Figure 3). We run Wu's algorithm for all possible non-cut vertex orders to reduce circuit size. For each permutation, select an optimal ordering, defined as one that results in the smallest CNOT circuit size, with depth as a tie-breaker.

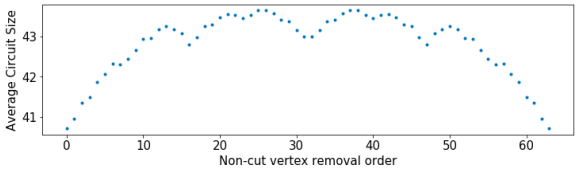

Figure 3: Effect of vertex removal order on average circuit size synthesized by ROWCOL for an 8Q path.

Since the ROWCOL algorithm reduces the problem size by one qubit in each iteration, it lends itself to hybridization with exact solvers. We thus terminate the heuristic early when the graph has only four qubits left (threshold chosen empirically), then call our CNOT-size-optimal solver on the reduced graph to finish synthesizing the circuit. (Note that SWAP-size-optimal is not an option since the sub-problems may not be permutations). We combine the circuits obtained by both methods to obtain the final result.

3.2 Results

We compare our approach to three previous CNOT-based methods: "Steiner-Gauss" by Kissinger et al. [8], "Linear-TF-Synth" by Gheorghiu et al. [7] and "ROWCOL" by Wu et al. [15], over all 8-qubit permutations on a path topology, as shown in Figure 4. Our ROWCOL-Hybrid algorithm with optimal vertex order achieves a smaller CNOT count than the size-optimal SWAP-based method for 88.8% of all permutations. At the same time, Kissinger et al. [8], Wu et al. [15], and Gheorghiu et al. [7] synthesize smaller CNOT

circuits than the optimal SWAP-based method for 12.5%, 54.8%, and 6.6% of all permutations, respectively.

We find this CNOT-based approach to be a surprisingly large improvement over using SWAPs, which naively seem to be naturally suited for permutations. While our approach primarily optimizes for circuit size, there is also an improvement in circuit depth compared to the three other algorithms.

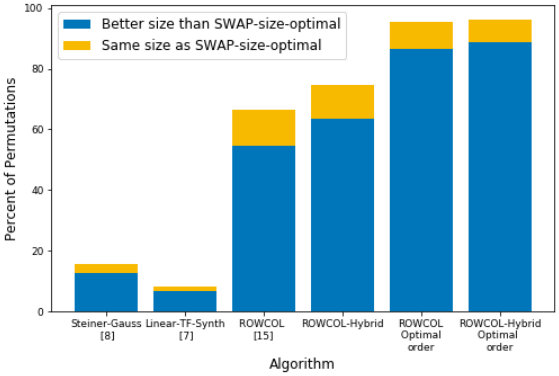

Figure 4: Comparison of how CNOT-size-optimizing methods improve upon SWAP-size-optimal methods (permutations on a path of 8). Our hybrid with optimal ordering approach achieves smaller CNOT count for 88.8% of all permutations.

4 DEPTH OPTIMIZATION: LR-SYNTH

Due to low qubit coherence times, circuit depth is often the most limiting factor in near-term quantum experiments. Existing depth-optimizing synthesis methods scale exponentially [12], assume full connectivity [4], are restricted to certain topologies [1, 9, 17], or assume non-standard primitive operations [2]. We propose a polynomial-time divide-and-conquer algorithm, LR-Synth, for synthesizing permutations on any limited-connectivity topology, optimizing for depth of SWAPs. Our algorithm may also be of independent interest for general network routing.

4.1 Algorithm Description

Given a permuted graph G_p and a target graph G_t, the goal is to convert G_p to G_t through a series of SWAPs, parallelizing as many SWAPs as possible. The high-level idea is as follows (see Figure 5): We partition G_p into two connected subgraphs as close in size as possible, then we move qubits to the correct half using maximal matchings. Once all qubits are on the correct half, we recursively call the algorithm on each half. Since the left and right halves are disjoint, their circuitry is parallelized on chip. The complete algorithm is given in Algorithm 1, where $D(G, a, b)$ denotes the shortest distance between vertices a and b in G.

The first step is partitioning G_p into two connected subgraphs, $leftG$ and $rightG$, as balanced as possible (step 1). This is the balanced connected 2-partition problem is NP-hard [5], so we use a heuristic to perform the partition. We use a simple heuristic that performs a depth first search starting from each node until half of the graph is traversed, removing the visited node each time to put into $leftG$, while keeping $rightG$—the part of the graph not traversed—connected. We find that this heuristic finds adequate splits for most topologies of interest. To maximize parallel routing across the partition, we select partitions resulting from removing vertex disjoint edges from G if possible (let $removedEdges$ be all edges in G_p but not in $leftG$ or $rightG$), because if two or more edges share a vertex, that vertex would become the bottleneck for routing vertices across the partition. In cases where no such partitions exist, we let $removedEdges$ be edges that form a maximal matching. Since trying every partition is infeasible for larger topologies, we sample up to S partitions.

Next, we assign each vertex that needs to be moved to the other half a path to take, specified by one of the removed edges (step 8). There are many possible heuristics for $assignPath$. Our heuristic assigns $v \in moveToRight$ to a path by choosing the (l, r) that minimizes

$$Cost = \max\{D(leftG, v, l), D(rightG, w, r)\} + n_{(l,r)}/2,$$

where w is the unassigned vertex in $moveToLeft$ closest to (l, r) and $n_{(l,r)}$ is the number of nodes assigned to (l, r). $D(leftG, v, l)$ and $D(rightG, w, r)$ are the number of swaps needed to move v and w respectively to the removed edge. Since this can be performed in parallel, we take the maximum distance. We add the term $n_{(l,r)}$ because v and w cannot be routed to the other side until qubits already assigned to the path (l, r) are routed, so $n_{(l,r)}$ penalizes moving too many qubits via one path.

After assigning paths, we iterate and add swaps until all qubits are routed to the correct side of the topology via their assigned paths (step 11). Let E be a set of potential edges to swap in a given iteration. At each iteration, we prioritize adding a swap by assigning it a weight of 1.3 if the swap moves vertex v to its final destination and all vertices on the path from v to a terminal node of G are already correctly positioned because making the swap effectively reduces the size of the topology we work with (step 12).

To explain steps 13 - 25, we consider, WLOG, $u \in moveToRight$, where u is routed across the removed edge (l, r). For any vertex $u \in moveToRight$, we would want to add a potential SWAP between u and v when $D(G, u, r) > D(G, v, r)$ (step 16), since this would bring u closer to the $rightG$. If $v \in moveToRight$ and $vertexToPath[v] = (l, r)$ and $D(G_t, u, rT) \leq D(G_t, rT)$, u is closer to the removed edge in the target graph, implying it should be routed to the right after v is routed, so we do not add (u, v) to E in this iteration (step 17). If no possible SWAPs are present in a given iteration, step 18 fixes this in the next iteration by swapping u and v if there is a neighbor b of v that is not in $moveToRight$ and b is closer than u to r, since then in a subsequent iteration, swapping u and b would move u closer to its target location.

If $v \in moveToRight$ and v's path is different from u, then we add (u, v) to E if the swap brings both u and v closer to their respective paths (step 19). If v is already on the correct side in $leftG$, then we add (u, v) to E because this would bring u closer to its destination without moving v to the wrong side (step 22). Finally, we swap u to the right side if $v \in moveToLeft$; otherwise, extra swaps would be needed to move v back to $rightG$ in a future iteration. We prioritize swaps that bring vertices to the correct side since this would allow more vertices to be routed via the same path, so we weight these swaps by 1.2 (step 23).

After steps 13 - 25, we find a maxMatching to minimize depth (step 26). In step 28, if G is a path or a ring topology, for each edge

(u, v) in G, if u and v are not in *moveToLeft* or *moveToRight* and their orders are flipped in G_t, we add (u, v) to *maxMatching* if adding the edge still results in a matching. This adds swaps that would otherwise occur later in the algorithm and takes advantage of earlier matchings to reduce depth.

Once *moveToRight* and *moveToLeft* are emptied, all vertices are on the correct half of the topology. After sampling S partitions, we choose the partition that empties *moveToRight* and *moveToLeft* in the least number of iterations (lowest depth) and use how close G's state is to G_t as a tiebreaker (step 32). Finally, we recursively call the algorithm on $leftG$ and $rightG$ (steps 33, 34). Since $leftG$ and $rightG$ contain disjoint vertices, the two calls are parallelized.

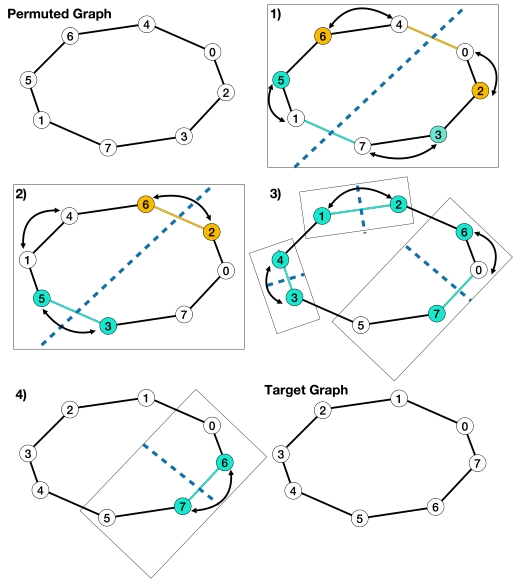

Figure 5: Algorithm steps on an 8Q ring topology. In steps 1 and 2, there are two possible paths to move a qubit from one side to another (yellow and blue edges). Qubits that must be routed to the other side have the same color as the path they are assigned, and the algorithm recurses in steps 3 and 4 until the target graph is achieved.

4.2 Results

We benchmark our algorithm by comparing LR-Synth to Tweedledum's SWAP-depth-optimal SAT solver [11, 12] for randomly sampled permutations on increasing qubit numbers. Due to the NP-complete nature of the SAT problem, the solver does not terminate quickly for certain permutations even at relatively small qubit numbers, although the specific number is topology dependent. To save computational time and resources, we run the SAT solver until the average circuit synthesis time exceeds 10 seconds on a typical personal computer. While LR-Synth can be applied to any limited connectivity topology, we perform our benchmarking on paths, trees, rings, and grids, commonly found on current physical devices. For line, tree and grid topologies, which have existing depth-optimizing heuristics, we also compare LR-Synth to them. For rings, there are many possible partitions for LR-Synth, so we compare the performance of randomly sampling a single partition versus selecting the best partition. We also hybridize LR-Synth with the SWAP-depth-optimal solver, analogous to the process for ROWCOL-Hybrid.

Algorithm 1: LR-Synth

Data: (i) Permuted graph G_p, (ii) Target graph G_t, (iii) Number of splits to sample S
Result: List of SWAPs taking G_p to G_t

1 leftGs, rightGs, removedEdges ← partitionGraph;
2 Swaps ← [ ];
3 **for** *split i = 1 to S* **do**
4 G ← G_p.copy; leftG ← leftGs[i]; rightG ← rightGs[i]; removedEdges ← removedEdges[i];
5 leftG_t, rightG_t ← corresponding left and right graphs of G_t;
6 moveToLeft ← {v : v ∈ rightG and v ∈ leftG_t}; ;
7 moveToRight ← {v : v ∈ leftG and v ∈ rightG_t};
8 vertexToPaths ← assignPath;
9 GL ← leftG ∪ removedEdges; GR ← rightG ∪ removedEdges;
10 lockL ← False; lockR ← False;
11 **while** *moveToRight + moveToLeft ≠ ∅* **do**
12 E = {(u, v, 1.3) : (u, v) ∈ G s.t. swapping u and v makes all vertices from u to terminal of G in correct positions};
13 **for** *(u, v) ∈ G s.t. u ∈ moveToRight + moveToLeft* **do**
14 (l, r) ← vertexToPaths[u];
15 (lT, rT) ← corresponding edge in G_t;
16 **if** *u ∈ moveToRight and D(GL, u, r) > D(GL, v, r)* **then**
17 **if** *v ∈ moveToRight and vertexToPath[v] = (l, r) and D(G_t, u, rT) ≤ D(G_t, v, rT)* **then** continue;
18 **if** *lockL and v ∈ moveToRight and vertexToPath[v] ≠ (l, r) and ∃ neighbor b of v s.t. b ∉ moveToRight and D(GL, u, r) > D(GL, b, r)* **then** E.add(u, v, 1); lockL ← False;
19 **if** *v ∈ moveToRight and vertexToPath[v] ≠ (l, r)* **then**
20 (a, b) ← vertexToPath[v];
21 **if** *D(GL, v, b) > D(GL, u, b)* **then** E.add(u, v, 1);
22 **else if** *u ≠ l* **then** E.add(u, v, 1);
23 **else if** *(u, v) = (l, r) and v in moveToLeft and vertexToPath[v] = (l, r)* **then** E.add(u, v, 1.2);
24 **if** *u ∈ moveToLeft and D(GR, u, l) > D(GR, v, l)* **then** Do similar logic as steps 16 - 23 ;
25 **end**
26 maxMatching ← maximum weight matching for E;
27 **if** *maxMatching = ∅* **then** lockL ← True; lockR ← True;
28 **if** *G is a path or ring* **then** addToMatching;
29 **for** *(u, v) ∈ maxMatching* **do** SWAP u and v in G;
30 Update moveToLeft and moveToRight;
31 **end**
32 Swaps ← SWAPs from best split;
33 **if** *size(leftG) > 1* **then** Swaps += LR-Synth(leftG, leftG_t);
34 **if** *size(rightG) > 1* **then** Swaps += LR-Synth(rightG, rightG_t);
35 **end**
36 **return** Swaps

4.2.1 Path. We compare LR-Synth to Kutin's odd-even transposition sort algorithm and SWAP-depth-optimal by sampling 100 random permutations from 4Q to 100Q (Figure 6a). We use Schoute's

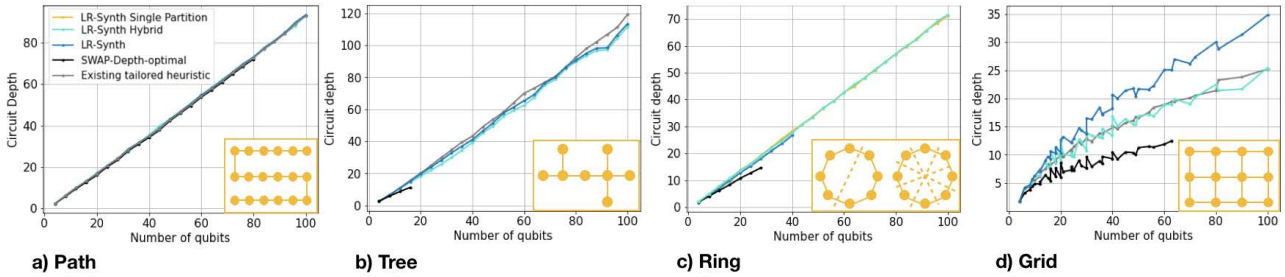

Figure 6: Depth comparison of four topologies for permutations synthesized using LR-Synth (hybrid, single, and all partitions), SWAP-depth-optimal, and tailored algorithms where one exists.

implementation of Kutin's algorithm [13]. Kutin and LR-Synth achieve same sized circuits and similar depth circuits, which are better in average size than SWAP-Depth-optimal and slightly worse in average depth than SWAP-Depth-optimal. SWAP-Depth-optimal's average circuit synthesis time exceeded 10 seconds at 84Q.

4.2.2 Tree. We compare LR-synth to Zhang's tree algorithm and SWAP-Depth-optimal by sampling 10 random trees for each tree size ranging from 4Q to 100Q For each tree topology, we sample 10 random permutations. We use Schoute's implementation of Zhang's algorithm [13]. SWAP-depth-optimal's average circuit synthesis time exceeds 10 seconds at 20Q. LR-Synth performs slightly better on average in terms of depth than Zhang.

4.2.3 Ring. We compare LR-Synth, sampling a single and all partitions, to SWAP-depth-optimal, averaging the results of 100 random permutations from 4Q to 100Q. The average circuit synthesis time of LR-Synth sampling all partitions exceeds 10 seconds at 44Q, compared to 28Q for SWAP-depth-optimal. The average circuit depths and sizes of sampling a single partition are comparable to those of all partitions, while the run time is significantly better.

4.2.4 Grid. Since grids have a well-defined structure, we partition G_p into two subgraphs by performing a cut midway along the x or y direction. We benchmark LR-Synth against SWAP-Depth-optimal and Alon's Cartesian algorithm for grid sizes (2, 2) to (10, 10). For each grid size, we sample 100 random permutations. Compared to other topologies, permutations on grids yield smaller average size and depth circuits. We observe this behavior for both LR-Synth and SWAP-Depth-optimal, and it is an intuitive result because there are more paths to route qubits that can be parallelized. We also note that the hybrid algorithm sees significant performance improvements on this topology.

5 CONCLUSION AND OPEN QUESTIONS

We proposed two algorithms for synthesis on limited-connectivity quantum computers. Using a modified ROWCOL algorithm hybridized with our optimal size solver for CNOTs, we show that permutations synthesized using CNOTs can significantly outperform the optimal circuit sizes achievable by SWAP-based methods. It follows that an important open question is whether there is a depth-optimizing algorithm for general topologies using CNOTs and how much improvement it can yield.

We proposed LR-Synth as a scalable depth-optimizing algorithm applicable to any topology. It achieves similar circuit depth as state-of-the-art heuristics tailored to paths, trees, and grids, and scales much better than optimal solvers. How much can LR-Synth be improved with better heuristics for partitioning the graph and choosing which path each qubit is routed through?

Finally, worst-case depth for permutation on a path remains an open question. We have shown a counter-example to the conjecture that reversal is the hardest permutation. What is the worst-case CNOT depth? We conjecture this to be $2n + O(1)$, rather than the known upper bound of $3n$.

REFERENCES

[1] N. Alon, F. R. K. Chung, and R. L. Graham. 1993. Routing permutations on graphs via matchings. In *STOC '93*.
[2] Aniruddha Bapat, Andrew M. Childs, Alexey V. Gorshkov, Samuel King, Eddie Schoute, and Hrishee Shastri. 2021. Quantum routing with fast reversals. *Quantum* 5 (Aug. 2021), 533. https://doi.org/10.22331/q-2021-08-31-533
[3] Ethan Bernstein and Umesh Vazirani. 1997. Quantum complexity theory. *SIAM Journal on computing* 26, 5 (1997), 1411–1473.
[4] Timothée Goubault de Brugière, Marc Baboulin, Benoît Valiron, Simon Martiel, and Cyril Allouche. 2021. Reducing the Depth of Linear Reversible Quantum Circuits. *IEEE Transactions on Quantum Engineering* 2 (2021), 1–22. https://doi.org/10.1109/TQE.2021.3091648
[5] Janka Chlebíková. 1996. Approximating the maximally balanced connected partition problem in graphs. *Inform. Process. Lett.* 60, 5 (1996), 225–230. https://doi.org/10.1016/S0020-0190(96)00175-5
[6] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. 2019. Validating quantum computers using randomized model circuits. *Physical Review A* 100, 3 (Sep 2019). https://doi.org/10.1103/physreva.100.032328
[7] Vlad Gheorghiu, Sarah Meng Li, Michele Mosca, and Priyanka Mukhopadhyay. 2021. Reducing the CNOT count for Clifford+T circuits on NISQ architectures. arXiv:2011.12191 [quant-ph]
[8] Aleks Kissinger and Arianne Meijer van de Griend. 2019. CNOT circuit extraction for topologically-constrained quantum memories. arXiv:1904.00633 [quant-ph]
[9] Samuel Kutin, David Moulton, and Lawren Smithline. 2007. Computation at a distance. *Chicago Journal of Theoretical Computer Science* 13 (02 2007).
[10] Ketan N Patel, Igor L Markov, and John P Hayes. 2008. Optimal synthesis of linear reversible circuits. *Quantum Inf. Comput.* 8, 3 (2008), 282–294.
[11] Bruno Schmitt. 2021. tweedledum. https://github.com/boschmitt/tweedledum.
[12] Bruno Schmitt, Mathias Soeken, and Giovanni De Micheli. 2020. Symbolic Algorithms for Token Swapping. In *2020 IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL)*. 28–33.
[13] Eddie Schoute. 2019. arct. https://gitlab.umiacs.umd.edu/amchilds/arct.
[14] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. https://doi.org/10.1109/SFCS.1994.365700
[15] Bujiao Wu, Xiaoyu He, Shuai Yang, Lifu Shou, Guojing Tian, Jialin Zhang, and Xiaoming Sun. 2019. Optimization of CNOT circuits on topological superconducting processors. arXiv:1910.14478 [quant-ph]
[16] Katsuhisa Yamanaka, Erik D Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. 2015. Swapping labeled tokens on graphs. *Theoretical Computer Science* 586 (2015), 81–94.
[17] Louxin Zhang. 1997. Optimal Bounds for Matching Routing on Trees. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*.